

## DEPARTMENT OF COMPUTER APPLICATIONS

### MODULE – 3

### PROCESS SYNCHRONIZATION & DEADLOCKS

#### TOPIC 1 – SYNCHRONIZATION: THE CRITICAL SECTION PROBLEM

##### Introduction

In multiprogramming systems, several processes may execute concurrently, often requiring access to shared data or hardware resources. When multiple processes access shared data simultaneously, the outcome of execution may depend on the order in which data access occurs — this situation is known as a race condition.

To prevent such inconsistencies, process execution must be properly synchronized. The section of code in which a process accesses shared data is termed the critical section.

The critical section problem involves designing a protocol that ensures processes cooperate correctly while accessing shared data without conflict.

The goal of synchronization is to maintain data consistency by allowing only one process at a time inside its critical section.

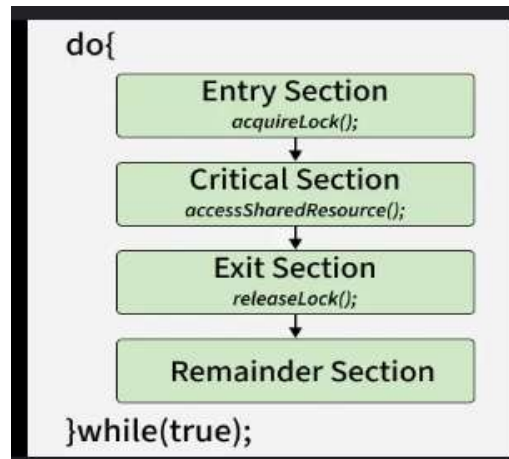
The operating system and hardware provide synchronization mechanisms such as locks, semaphores, monitors, and atomic instructions.

A correct solution must satisfy three essential requirements: mutual exclusion, progress, and bounded waiting.

Synchronization is fundamental to the design of reliable multiprocessor systems and ensures that shared resources are used safely, efficiently, and consistently.

Proper synchronization prevents data corruption, guarantees predictable execution, and forms the basis for interprocess communication and multithreading.

## DEPARTMENT OF COMPUTER APPLICATIONS



### Explanation

Every process in a concurrent system alternates between executing in its critical section and in its remainder section.

To ensure correct behavior, the operating system must implement synchronization such that:

1. Mutual Exclusion: No two processes can execute their critical sections simultaneously.
2. Progress: If no process is in its critical section, and one or more processes wish to enter, selection of the next process cannot be postponed indefinitely.
3. Bounded Waiting: A limit must exist on the number of times other processes can enter their critical sections before a waiting process gets access.

The critical section problem can be addressed through software and hardware solutions.

- Software solutions use shared variables and flags to coordinate access between processes, e.g., Peterson's Algorithm.
- Hardware solutions rely on special atomic machine instructions like Test-and-Set or Compare-and-Swap, which execute indivisibly.
- High-level synchronization mechanisms such as semaphores and monitors provide structured and efficient ways to handle critical sections.

## DEPARTMENT OF COMPUTER APPLICATIONS

When a process enters its critical section, it locks access to shared resources. Once it exits, it releases the lock so other processes can enter.

Improper synchronization may cause race conditions, where multiple processes attempt to modify shared variables simultaneously, producing unpredictable results.

Thus, the OS enforces rules to ensure safe entry and exit from critical sections using atomic operations and scheduling control. Synchronization primitives provided by modern operating systems (mutexes, semaphores, condition variables) ensure that all processes execute in a controlled and predictable manner.

### Example

Consider two processes that share a variable counter. Both processes increment the counter by one.

Without synchronization, both might read the same value, increment it, and write back the same result — losing one update.

Using a semaphore mutex initialized to 1, mutual exclusion can be achieved:

```
wait(mutex);  
counter = counter + 1;  
signal(mutex);
```

Only one process at a time executes the critical section, ensuring that the counter is incremented correctly.

### Conclusion

Synchronization ensures orderly execution of concurrent processes and prevents data inconsistency caused by race conditions.

The critical section problem lies at the core of process coordination.

By satisfying mutual exclusion, progress, and bounded waiting, the OS guarantees predictable and reliable execution in multiprogramming environments.

## DEPARTMENT OF COMPUTER APPLICATIONS

### TOPIC 2 – PETERSON'S SOLUTION

#### Introduction (20 lines)

Peterson's solution is a classic software-based algorithm that provides a simple and elegant way to achieve mutual exclusion between two concurrently executing processes that share data. It uses only ordinary load and store machine instructions—no special hardware support is needed—and still satisfies the three requirements of the critical-section problem: mutual exclusion, progress, and bounded waiting.

- Gary Peterson proposed this algorithm to demonstrate that proper synchronization could be achieved entirely in software when only two processes are involved.
- Each process alternates between a critical section, where it manipulates shared data, and a remainder section, where it performs other operations.
- To coordinate access, the processes communicate through two shared variables: an array `flag[2]` that indicates intent to enter the critical section, and an integer variable `turn` that indicates whose turn it is.
- Before entering the critical section, a process raises its flag and sets the turn to the other process, voluntarily giving the other process the opportunity to enter first if it also wants to.
- Only when the other process is not interested or has finished does the waiting process proceed into its own critical section.
- Peterson's solution is conceptually important because it proves that mutual exclusion and progress can be achieved without busy waiting loops that depend on complex hardware instructions.
- It forms the theoretical foundation for many synchronization mechanisms used in operating systems and concurrent programming.
- Although it is primarily of educational value, it clearly illustrates the logic of coordination between processes.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Explanation

Peterson's solution assumes that load and store instructions execute atomically—that is, no other instruction can interleave while a single assignment statement is executed.

Two shared data items coordinate the entry protocol:

1. `flag[i]` — set to TRUE when process  $P_i$  wants to enter its critical section.
2. `turn` — indicates whose turn it is to enter the critical section when both want in simultaneously.

Algorithm Steps (for Process  $P_i$ ):

1. Indicate interest: `flag[i] = TRUE;`
2. Give priority to the other process: `turn = j;`
3. Wait while the other process is interested and it's the other's turn: `while(flag[j] && turn==j);`
4. Execute the critical section.
5. Reset `flag[i] = FALSE;` on leaving.

### Why it works:

- Mutual Exclusion: Both processes cannot be inside the critical section simultaneously because at most one of the conditions `flag[j] && turn==j` can be false at any given time.
- Progress: If one process is not interested (`flag[j]==FALSE`), the other enters immediately.
- Bounded Waiting: Each process gets a fair chance because the `turn` variable alternates between them.

Peterson's solution is elegant but limited to two processes. Extending it to  $n$  processes is possible but complicated and inefficient.

Modern processors often reorder memory accesses, so in real hardware the algorithm may require memory barriers to function correctly.

Still, it remains a fundamental theoretical example illustrating that synchronization can be

## DEPARTMENT OF COMPUTER APPLICATIONS

achieved purely in software without special machine instructions.

The key insight is voluntary yielding: each process explicitly gives the other process a chance to enter first, preventing deadlock and ensuring fairness.

### Example

- Consider two threads incrementing a shared variable counter.
- Each thread uses the Peterson protocol before the increment operation.
- If both attempt to enter the critical section, the turn variable ensures that only one proceeds while the other waits.
- Once the first thread completes its increment and sets its flag to FALSE, the waiting thread observes that the other is no longer interested and proceeds safely.

*Thus, every increment is performed exactly once, maintaining correct results without race conditions.*

### Conclusion

Peterson's solution demonstrates that software alone can provide correct synchronization between concurrent processes.

It guarantees mutual exclusion, progress, and bounded waiting using minimal shared data. Although rarely used in modern systems, its conceptual simplicity makes it invaluable for understanding the fundamentals of process synchronization.

It serves as the cornerstone example of how logic and discipline can replace complex hardware in achieving safe concurrency.

## DEPARTMENT OF COMPUTER APPLICATIONS

### TOPIC 3 – SYNCHRONIZATION HARDWARE

#### Introduction

Software solutions like Peterson's algorithm work well in theory, but they are limited to a small number of processes and depend on strict sequential consistency of memory operations.

In real multiprocessor systems, the operating system relies heavily on hardware-based synchronization mechanisms to coordinate concurrent processes efficiently.

Modern CPUs provide atomic instructions — operations that are executed as a single, uninterruptible step — allowing safe modification of shared variables without interference from other processes or threads.

These atomic primitives are implemented directly in hardware, ensuring that no other instruction can access the same memory location until the operation completes.

Common examples include Test-and-Set, Compare-and-Swap (CAS), Exchange, and Fetch-and-Add.

These instructions form the foundation for implementing high-level synchronization constructs such as locks, mutexes, and semaphores.

Hardware-based synchronization reduces overhead and prevents race conditions at the hardware level itself, making it more reliable and efficient than pure software techniques.

By combining these atomic operations with OS-level data structures, processes can safely enter and exit their critical sections without busy waiting or inconsistent behavior.

However, if not carefully designed, these methods can still lead to issues such as priority inversion, busy waiting, or starvation.

Therefore, while synchronization hardware provides the building blocks, it must be combined with proper software control for fairness and efficiency.

#### Explanation

Hardware synchronization provides primitive operations that allow processes to implement mutual exclusion without requiring operating system intervention for every entry and exit of a critical section.

## DEPARTMENT OF COMPUTER APPLICATIONS

These instructions are performed atomically, meaning they are executed in one uninterruptible machine cycle.

Let's examine the most common types:

### 1. Test-and-Set (TAS) Instruction:

- The instruction reads the value of a memory location and sets it to TRUE in one atomic step.
- Example in pseudo-code:

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

- If the previous value was FALSE, the process gains access to the critical section; otherwise, it loops until the value becomes FALSE again.
- This is used to implement locks or spinlocks.

### 2. Compare-and-Swap (CAS):

- Compares the contents of a memory location with an expected value and, only if they match, swaps it with a new value atomically.
- Pseudo-code:

```
int CompareAndSwap(int *value, int expected, int new_value) {  
    int temp = *value;  
    if (temp == expected)  
        *value = new_value;  
    return temp;  
}
```



## DEPARTMENT OF COMPUTER APPLICATIONS

- CAS is more flexible than Test-and-Set and is widely supported in modern processors.
- 3. Exchange (XCHG):
  - Atomically swaps the contents of two memory locations. Used to implement low-level locking mechanisms.
- 4. Fetch-and-Add:
  - Atomically increments a memory variable and returns its old value. Useful for implementing counting semaphores.

These operations ensure that multiple processes or threads accessing shared data structures (like queues, tables, or buffers) can do so safely.

However, busy waiting (also called spinlocking) can waste CPU cycles when a process loops repeatedly, waiting for a lock to become available.

To mitigate this, many systems combine hardware synchronization with scheduling — e.g., the OS may block a process if it waits too long.

Hardware primitives are simple but powerful and are the basis for all modern concurrent control mechanisms in kernels and real-time systems.

### Example

A simple lock can be implemented using the Test-and-Set instruction:

```
do {  
    while (TestAndSet(&lock)); // busy wait until lock is free  
    /* Critical Section */  
    lock = FALSE;           // release the lock  
    /* Remainder Section */  
} while (TRUE);
```

## DEPARTMENT OF COMPUTER APPLICATIONS

Here, the atomicity of Test-and-Set ensures that only one process at a time can enter the critical section.

### Conclusion

Hardware synchronization provides the essential atomic primitives that guarantee safe concurrent access to shared resources.

Operations like Test-and-Set and Compare-and-Swap eliminate race conditions at the processor level.

These mechanisms are fundamental for building higher-level synchronization tools such as semaphores, monitors, and spinlocks.

While hardware support ensures speed and reliability, it must be managed carefully to avoid CPU wastage through busy waiting.

## TOPIC 4 – SEMAPHORES

### Introduction

In concurrent programming, semaphores are one of the most powerful and widely used synchronization mechanisms provided by the operating system.

Introduced by Edsger Dijkstra in 1965, a semaphore is an integer variable used to control access to shared resources by multiple processes in a concurrent system.

Semaphores solve the critical section problem by signaling and mutual exclusion rather than busy waiting.

Unlike hardware primitives that rely on processor-specific instructions, semaphores offer a portable, high-level abstraction to coordinate process execution.

They are implemented by the OS using atomic operations to avoid race conditions.

Semaphores can be binary (mutex) or counting, depending on whether they control access to a single or multiple instances of a resource.

Processes use two standard atomic operations on semaphores: wait() (also known as P or down operation) and signal() (also known as V or up operation).

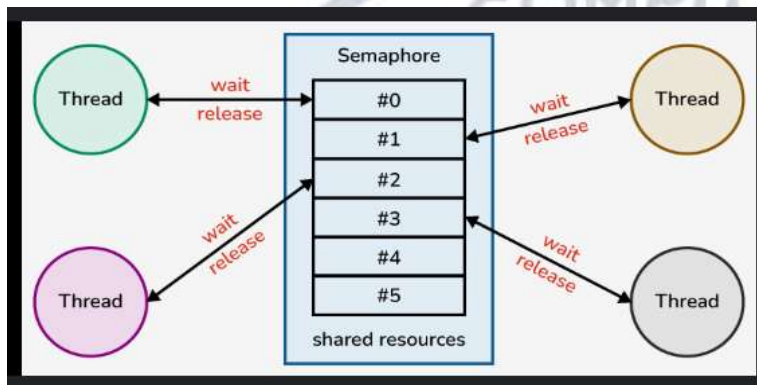
## DEPARTMENT OF COMPUTER APPLICATIONS

When a process performs a wait() on a semaphore whose value is positive, it decrements it and continues; if it is zero, the process is blocked until another process executes signal().

This blocking and signaling mechanism eliminates CPU wastage due to busy waiting.

Semaphores are fundamental for solving classical synchronization problems such as producer-consumer, readers-writers, and dining philosophers.

Diagram (Semaphore Concept)



### Explanation

A semaphore is a synchronization variable that can take nonnegative integer values.

It is accessed through two atomic operations:

wait(S) (also called P or down operation):

```
wait(S) {
    while (S <= 0) ; // busy wait
    S--;
}
```

If the semaphore value is positive, it decrements it and allows the process to proceed; otherwise, it waits until the value becomes positive.

signal(S) (also called V or up operation):

## DEPARTMENT OF COMPUTER APPLICATIONS

```
signal(S) {  
    S++;  
}
```

It increments the semaphore value, possibly waking up a waiting process.

To avoid busy waiting, semaphores in modern systems are implemented using process blocking.

When a process executes `wait(S)` and finds `S == 0`, it is placed into a waiting queue associated with the semaphore, and the CPU is allocated to another process.

When another process calls `signal(S)`, one of the waiting processes is resumed.

Types of Semaphores:

- Binary Semaphore (Mutex):  
Has only two values (0 and 1). It ensures mutual exclusion in critical sections.
- Counting Semaphore:  
Initialized to the number of available instances of a resource. Used when multiple identical resources exist (e.g., several printers).

Semaphore Implementation in OS:

The operating system maintains a record structure for each semaphore containing:

- An integer value
- A waiting queue (for blocked processes)

Semaphores can be used to enforce synchronization between independent processes. For example, in a producer-consumer system, semaphores ensure that the producer does not overwrite data in a full buffer and the consumer does not consume from an empty buffer.

However, semaphores must be used carefully because improper use can lead to deadlocks (if processes wait indefinitely) or priority inversion (if a high-priority process is blocked by a low-priority one holding the semaphore).



## DEPARTMENT OF COMPUTER APPLICATIONS

Despite these issues, semaphores remain the foundation for higher-level synchronization constructs such as monitors and condition variables in modern systems.

### Example

Producer–Consumer Problem using Semaphores:

```
semaphore mutex = 1;  
semaphore full = 0;  
semaphore empty = n;
```

Producer:

```
do {  
    wait(empty);  
    wait(mutex);  
    // add item to buffer  
    signal(mutex);  
    signal(full);  
} while(TRUE);
```

Consumer:

```
do {  
    wait(full);  
    wait(mutex);  
    // remove item from buffer  
    signal(mutex);  
    signal(empty);  
} while(TRUE);
```

Here, mutex ensures mutual exclusion, while full and empty synchronize the buffer's state.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Conclusion

Semaphores are versatile and efficient synchronization tools for managing concurrent processes and shared resources.

They prevent race conditions by controlling process access to critical sections.

Although simple in concept, improper use may cause deadlocks or indefinite blocking.

When used correctly, semaphores form the backbone of reliable process synchronization in modern operating systems.

### Topic 4 – Semaphores

#### Introduction

A semaphore is a synchronization variable introduced by Edsger Dijkstra to manage concurrent access to shared resources in multiprogramming systems.

It is an integer variable that can only be accessed through two atomic operations—wait (P) and signal (V).

Semaphores help avoid race conditions by coordinating process execution order when several processes share common data.

They are more reliable and efficient than pure software solutions because they eliminate the need for continuous busy waiting.

A semaphore's value represents the number of available units of a resource.

When a process executes wait, the value decreases; when it executes signal, the value increases.

If the value becomes negative, processes are placed in a waiting queue until another process signals the semaphore.

Semaphores are implemented inside the kernel using atomic machine instructions.

They come in two main types: binary semaphores (mutex) for mutual exclusion and counting semaphores for multiple identical resources.

They are widely used in process synchronization, particularly in producer–consumer, readers–writers, and dining philosophers problems.

## DEPARTMENT OF COMPUTER APPLICATIONS

By using semaphores, the operating system ensures orderly execution and consistent data sharing among concurrent processes.

### Explanation

Semaphores provide a simple mechanism for controlling access to shared resources in concurrent systems.

Two atomic primitives manage them:

1. wait(S): decreases the semaphore value. If it becomes  $< 0$ , the process is blocked and placed in a waiting queue.
2. signal(S): increases the semaphore value. If the value  $\leq 0$ , one waiting process is awakened.

The OS ensures both operations are indivisible, preventing race conditions.

A binary semaphore has only 0 or 1 and is used for mutual exclusion (mutex).

A counting semaphore can have any non-negative value and tracks multiple resource instances.

Unlike busy-waiting locks, semaphore implementations in the kernel suspend blocked processes, thus saving CPU time.

They are the building blocks for higher-level synchronization mechanisms such as monitors and condition variables.

If semaphores are misused—e.g., a process forgets to signal—they can cause deadlocks or starvation, so careful design is necessary.

### Example

Producer–Consumer with Semaphores:

semaphore mutex = 1;

semaphore empty = n, full = 0;

Producer:

## DEPARTMENT OF COMPUTER APPLICATIONS

```
do {  
    wait(empty); wait(mutex);  
    produce_item(); insert_item();  
    signal(mutex); signal(full);  
} while(TRUE);
```

Consumer:

```
do {  
    wait(full); wait(mutex);  
    remove_item(); consume_item();  
    signal(mutex); signal(empty);  
} while(TRUE);
```

Here, mutex ensures mutual exclusion; empty and full maintain buffer synchronization.

### Conclusion

Semaphores are fundamental synchronization primitives that guarantee orderly access to shared resources.

They combine simplicity with efficiency, avoiding race conditions and busy waiting.

Both binary and counting semaphores are used widely in operating systems and concurrent applications.

Proper use of semaphores leads to consistent, deadlock-free process coordination and stable system performance.



## DEPARTMENT OF COMPUTER APPLICATIONS

### Topic 5 – Classical Problems of Synchronization

#### Introduction

Classical synchronization problems are standard examples that illustrate how multiple processes can cooperate and share resources without conflicts.

They serve as practical applications of synchronization primitives like semaphores, monitors, and mutexes.

The most common ones are the Bounded Buffer (Producer–Consumer) problem, the Readers–Writers problem, and the Dining Philosophers problem.

Each demonstrates how improper coordination leads to race conditions, deadlocks, or starvation, and how correct semaphore usage prevents them.

These models are fundamental to understanding how the operating system schedules and synchronizes processes that share data structures such as buffers, databases, and files.

By studying these problems, one learns to design concurrent systems that guarantee mutual exclusion, avoid deadlocks, and ensure fairness among competing processes.

#### Explanation

##### 1. Bounded Buffer (Producer–Consumer) Problem:

A fixed-size buffer connects a producer that generates data and a consumer that uses it. The producer must wait when the buffer is full, and the consumer must wait when it is empty.

Three semaphores are used:

- mutex (initial = 1) ensures mutual exclusion in buffer access.
- empty (initial = n) counts empty slots.
- full (initial = 0) counts filled slots.

The algorithm guarantees that producers and consumers never access the buffer simultaneously or exceed its bounds.

##### 2. Readers–Writers Problem:

A shared data structure, such as a database, can be read by multiple readers

## DEPARTMENT OF COMPUTER APPLICATIONS

simultaneously but written by only one writer at a time.

The goal is to allow maximum concurrency (multiple readers) while ensuring writers exclusive access.

Typical semaphore solution:

- mutex controls access to the read-count variable.
- rw\_mutex ensures that either writers or readers are in the critical section.
- Readers increment readcount; if it becomes 1, they lock rw\_mutex.
- Writers wait on rw\_mutex before modifying the data.

This coordination prevents readers and writers from entering simultaneously.

### 3. Dining Philosophers Problem:

Five philosophers sit around a circular table; each alternates between thinking and eating.

To eat, a philosopher must pick up the chopsticks to the left and right.

If all philosophers pick up their left chopstick simultaneously, a deadlock occurs.

Semaphore-based solutions limit simultaneous eating to avoid circular waiting.

One method uses a mutex for each chopstick and ensures that a philosopher picks up both or none.

Another approach uses a monitor to enforce that at most four philosophers eat concurrently.

The problem models resource allocation in multiprocessor systems where circular waiting must be prevented.

These classic problems demonstrate the effectiveness of semaphores in preventing race conditions and maintaining process synchronization.

They form the conceptual foundation for modern concurrency control in kernels, databases, and distributed systems.

## Conclusion

The classical synchronization problems provide practical insight into designing systems that manage shared resources safely.

## DEPARTMENT OF COMPUTER APPLICATIONS

Solutions using semaphores or monitors illustrate essential principles of mutual exclusion, bounded waiting, and avoidance of deadlock or starvation.

Mastering these patterns prepares developers to build correct and efficient concurrent programs in real operating systems.

### Topic 6 – Deadlocks: System Model

#### Introduction

In a multiprogramming environment, multiple processes often compete for finite system resources such as CPU cycles, memory, files, and I/O devices.

When processes hold some resources while waiting for others held by other processes, a deadlock may occur.

A deadlock is a situation where a set of processes are permanently blocked because each process is waiting for a resource that another process holds.

To describe and analyze deadlocks, operating systems use a resource-allocation system model.

In this model, resources are divided into types (like printers, memory blocks, etc.), and each resource type has one or more identical instances.

Each process follows a resource usage pattern: request → use → release.

A process must request a resource before using it and release it after use.

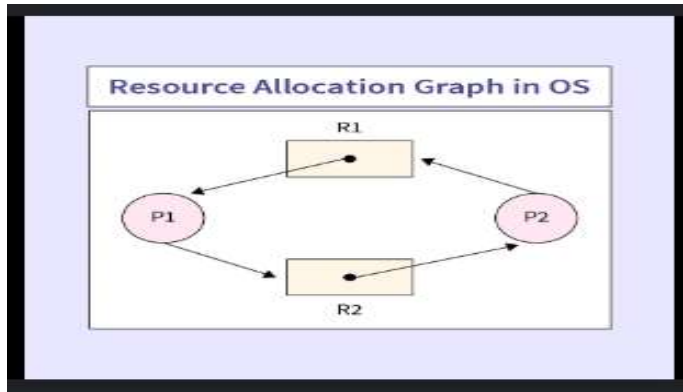
If the requested resource is unavailable, the requesting process must wait.

Deadlocks arise when each process in a set holds one resource and waits for another, creating a circular chain of waiting processes.

Understanding this system model helps identify conditions that lead to deadlocks and forms the foundation for methods to prevent or avoid them.

#### Diagram (Resource Allocation Graph)

## DEPARTMENT OF COMPUTER APPLICATIONS



A cycle in the graph indicates a possible deadlock.

### Explanation

The system model consists of:

- A set of processes:  $P = \{P_1, P_2, \dots, P_n\}$
- A set of resource types:  $R = \{R_1, R_2, \dots, R_m\}$

Each resource type  $R_i$  has  $W_i$  instances (e.g., 2 printers, 4 tape drives).

Processes request resources before using them and release them afterward.

At any time, a resource may be in one of two states: free (available) or allocated.

The operating system maintains a Resource Allocation Graph (RAG) that visually represents processes and resources.

Edges in the graph:

- Request edge ( $P_i \rightarrow R_j$ ): process  $P_i$  is waiting for resource  $R_j$ .
- Assignment edge ( $R_j \rightarrow P_i$ ): resource  $R_j$  is assigned to process  $P_i$ .

A cycle in the RAG indicates a deadlock in systems with single instances of each resource type.

For systems with multiple instances, a cycle may indicate the possibility of a deadlock but not certainty.



## DEPARTMENT OF COMPUTER APPLICATIONS

This system model simplifies understanding of process-resource relationships and aids in applying algorithms for detection, prevention, and avoidance.

### Example

Suppose:

- Process P1 requests and holds R1, then requests R2.
- Process P2 holds R2 and requests R1.

This creates a cycle  $P1 \rightarrow R2 \rightarrow P2 \rightarrow R1 \rightarrow P1$ , representing a deadlock.

### Conclusion

The system model abstracts how processes and resources interact, helping to analyze and detect deadlocks.

It provides a graphical foundation for methods like prevention, avoidance, and detection.

By maintaining proper resource allocation, operating systems can minimize deadlock situations.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Topic 7 – Deadlock Characterization

#### Introduction

A deadlock is not a random event; it occurs only when specific conditions hold simultaneously. The characterization of deadlocks defines these conditions, allowing operating systems to identify or prevent them.

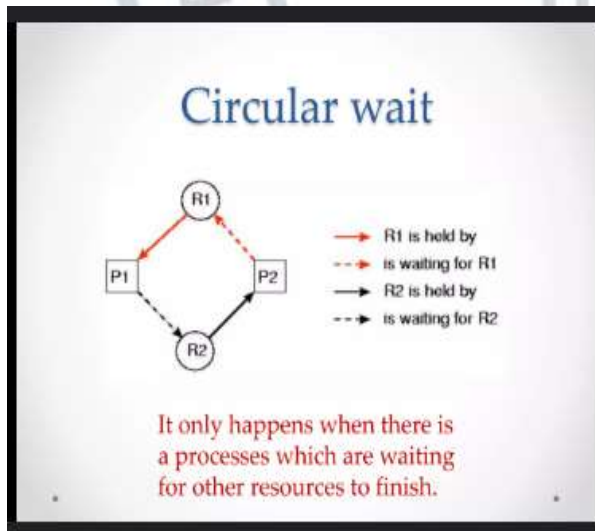
According to the system model, four necessary conditions must exist for a deadlock to occur. If even one of them is removed, deadlocks cannot arise.

These four conditions are Mutual Exclusion, Hold and Wait, No Preemption, and Circular Wait. Understanding these conditions is crucial because all deadlock-handling strategies are based on preventing or avoiding one or more of them.

They apply universally to all resource-sharing environments, including processes, threads, databases, and distributed systems.

This characterization also aids in constructing algorithms that monitor or break these conditions dynamically.

#### Diagram (Cycle Representation)



This circular wait represents all conditions for deadlock existence.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Explanation

#### 1. Mutual Exclusion:

At least one resource must be held in a non-shareable mode; only one process can use it at a time.

If another process requests the resource, it must wait until it's released.

Example: printers or disk drives.

#### 2. Hold and Wait:

A process is holding at least one resource while waiting to acquire additional ones.

Example: a process holds memory blocks and waits for I/O devices.

#### 3. No Preemption:

Resources cannot be forcibly removed from processes holding them.

They must be released voluntarily after use.

#### 4. Circular Wait:

There exists a circular chain of processes where each process holds a resource needed by the next.

Example: P1 holds R1 and waits for R2; P2 holds R2 and waits for R1.

All four conditions must hold simultaneously for a deadlock to occur.

By denying any one of them, the system can prevent deadlocks.

### Example

Consider two processes, P1 and P2, and two resources, R1 and R2.

- P1 holds R1 and requests R2.
- P2 holds R2 and requests R1.

All four deadlock conditions are satisfied: mutual exclusion, hold-and-wait, no preemption, and circular wait.

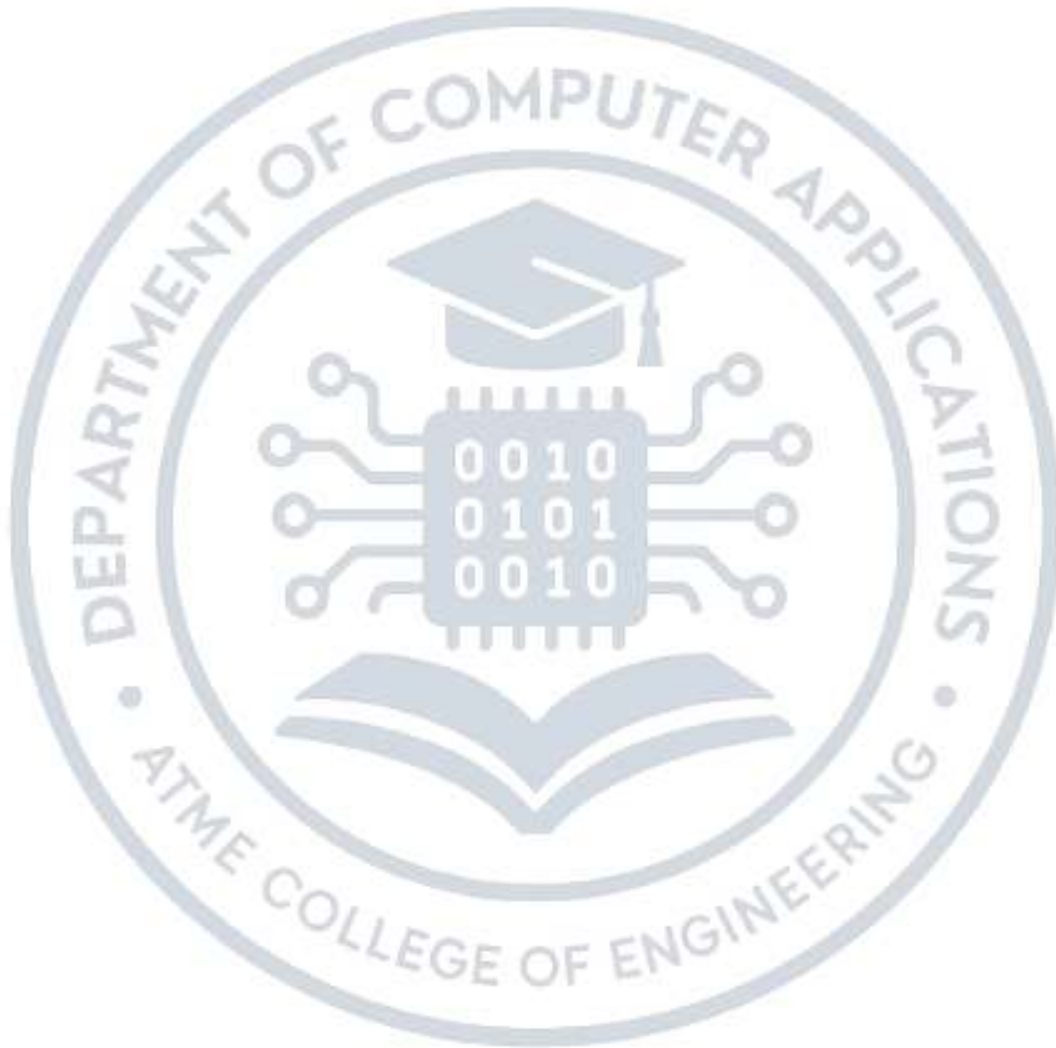
## DEPARTMENT OF COMPUTER APPLICATIONS

### Conclusion

Deadlock characterization explains the root causes of deadlock formation.

Recognizing the four necessary conditions allows the operating system to design prevention and avoidance techniques.

By breaking one condition, deadlocks can be effectively avoided or minimized.





## DEPARTMENT OF COMPUTER APPLICATIONS

### Topic 8 – Methods for Handling Deadlocks

#### Introduction

Once deadlocks are understood and characterized, the operating system must decide how to handle them.

There are four major strategies for dealing with deadlocks: Prevention, Avoidance, Detection and Recovery, and Ignoring the Problem.

Each method has its advantages and trade-offs in complexity, overhead, and efficiency.

Deadlock handling aims to ensure that the system continues to operate even when resource contention is high.

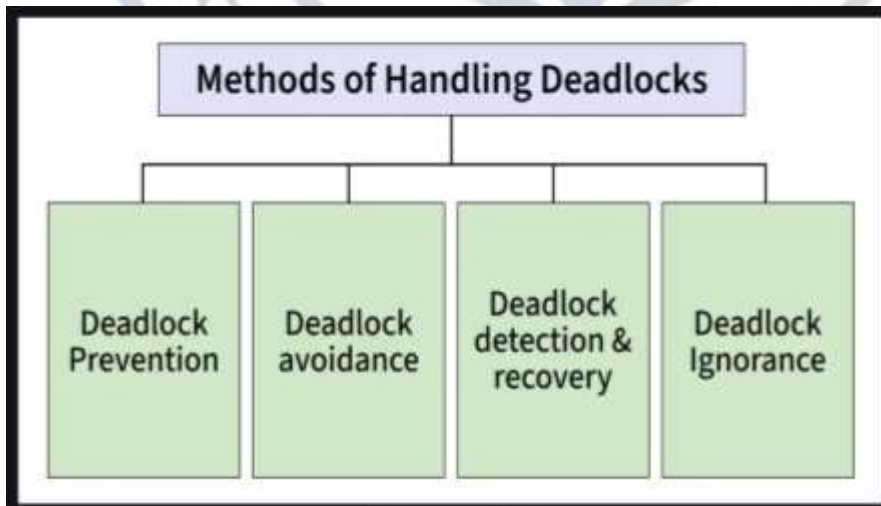
Some operating systems, like UNIX, take no special action against deadlocks, assuming they occur rarely and can be handled manually.

Others employ algorithms to prevent or detect deadlocks dynamically.

The method chosen depends on system design goals such as safety, performance, and resource utilization.

Efficient deadlock handling improves throughput and minimizes process starvation.

#### Diagram (Deadlock Handling Strategies)



## DEPARTMENT OF COMPUTER APPLICATIONS

### Explanation

#### 1. **Deadlock Prevention:**

- Ensures that at least one of the four necessary conditions never holds.
- For example, prohibit hold-and-wait by requiring all resources to be requested at once.
- This approach is simple but may reduce system resource utilization.

#### 2. **Deadlock Avoidance:**

- Allows resource requests dynamically but checks for safety before allocation.
- Uses algorithms like the **Banker's Algorithm** to ensure the system never enters an unsafe state.
- Requires knowledge of each process's maximum future requests.

#### 3. **Deadlock Detection and Recovery:**

- Allows deadlocks to occur but detects them using resource allocation graphs or matrices.
- Once detected, recovery is done by aborting processes or preempting resources.
- This approach suits systems where deadlocks are infrequent but costly to prevent.

#### 4. **Ignoring Deadlocks:**

- Used by most general-purpose systems like UNIX and Windows.
- Assumes deadlocks are rare and lets users terminate or restart processes manually.
- Simplifies design but risks indefinite blocking.

Each method must balance resource utilization, complexity, and safety.

Real systems often combine multiple strategies to minimize deadlock impact.

### Example

In a banking system, each process represents a customer who requests loans (resources).

The Banker's Algorithm (avoidance) ensures that the bank never allocates funds that lead to an unsafe state.

## DEPARTMENT OF COMPUTER APPLICATIONS

If a deadlock occurs (detection), the bank may recover by canceling some transactions or preempting resources.

### Conclusion

Deadlock handling is essential for maintaining system stability in multiprogrammed environments.

The operating system can prevent, avoid, or detect and recover from deadlocks based on performance needs.

Understanding these strategies enables effective resource management and reliable system operation.

### Topic 9 – Deadlock Prevention

#### Introduction

**Deadlock prevention** is a strategy that ensures the system is structured so that at least one of the four necessary conditions for deadlock (mutual exclusion, hold and wait, no preemption, circular wait) can never occur.

By breaking one or more of these conditions, the operating system can guarantee that deadlocks are structurally impossible.

The key idea is to **restrict resource requests** in ways that make circular or indefinite waiting impossible.

However, this often reduces concurrency and may lower resource utilization.

Prevention mechanisms are static; they are applied before a deadlock occurs rather than during execution.

Because of their simplicity, prevention techniques are mainly used in systems where predictability and reliability are more important than performance.

For example, embedded or real-time systems may favor prevention to ensure guaranteed execution.

## DEPARTMENT OF COMPUTER APPLICATIONS

Deadlock Conditions

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

↓ ↓ ↓ ↓

Break **any one** to prevent

Deadlock

### Explanation

To prevent deadlocks, at least one of the four necessary conditions is denied:

#### 1. **Mutual Exclusion:**

- Make resources shareable if possible (e.g., read-only files).
- Not applicable for inherently non-shareable resources like printers.

#### 2. **Hold and Wait:**

- Require processes to request all resources at once before execution.
- Alternatively, require a process to release all resources before requesting new ones.
- This approach avoids hold-and-wait but can lead to low utilization or starvation.

#### 3. **No Preemption:**

- Allow resources to be preempted if the requesting process cannot be satisfied.
- For example, if a process holding some resources requests another unavailable one, its currently held resources are released for others to use.
- After becoming available, the process reattempts its request.

#### 4. **Circular Wait:**



## DEPARTMENT OF COMPUTER APPLICATIONS

- Impose a total ordering of resource types and require processes to request resources in that order.
- Example: assign an increasing order (printer < disk < tape).
- This eliminates the possibility of circular waiting.

Each method reduces the likelihood of deadlocks but may affect system performance.

In practice, combining multiple approaches yields more balanced performance.

### Example

In a print-disk system:

- Processes must first request the printer and then the disk (following a fixed order).
- If every process follows this global ordering, circular wait is impossible, and hence deadlocks are prevented.

### Conclusion

Deadlock prevention removes the structural possibility of deadlocks by restricting process behavior.

Although effective, it may reduce system throughput and flexibility.

In systems where safety is critical, prevention remains a preferred and predictable solution.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Topic 10 – Deadlock Avoidance

#### Introduction

While prevention eliminates the possibility of deadlocks by imposing rigid rules, **deadlock avoidance** takes a dynamic approach.

Here, the system allows resource requests as long as granting them does not lead to an unsafe state.

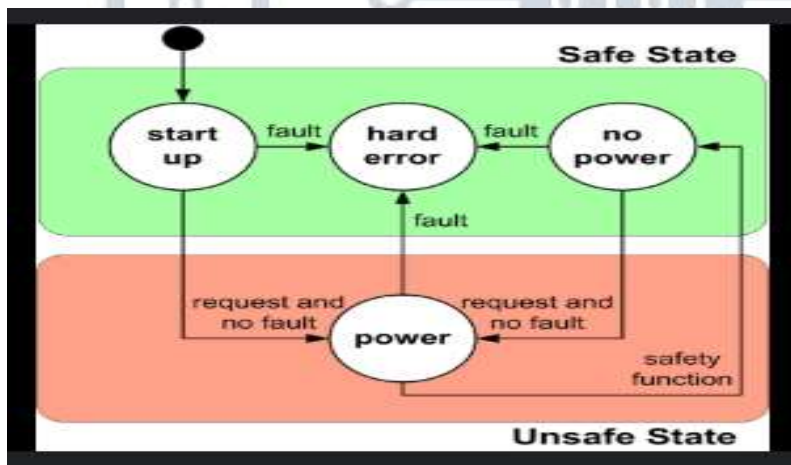
The operating system continuously analyzes resource allocation to ensure the system remains in a **safe state**, where at least one process sequence can complete successfully.

If a requested allocation could lead to a deadlock, it is temporarily denied.

The most famous avoidance algorithm is the **Banker's Algorithm**, proposed by Dijkstra.

This approach requires prior knowledge of each process's maximum resource requirements.

Although more flexible than prevention, it introduces computational overhead due to frequent safety checks.



#### Explanation

##### 1. Safe State:

A system is in a safe state if it can allocate resources to each process in some order such that every process can complete execution even if all request their maximum resources

## DEPARTMENT OF COMPUTER APPLICATIONS

simultaneously.

A **safe sequence** is the order of processes that guarantees this property.

### 2. Unsafe State:

A state that may lead to deadlock if future requests occur.

Not all unsafe states cause deadlocks, but deadlocks always originate from unsafe states.

### 3. Banker's Algorithm (for multiple resource types):

- Modeled after a banker lending resources (money) to customers (processes).
- Each process declares its **maximum** resource need.
- The OS checks before allocation whether granting the request keeps the system in a safe state.
- If yes → resources are allocated; otherwise → the process waits.

Data structures used:

- **Available:** Vector of available resources.
- **Max:** Maximum demand of each process.
- **Allocation:** Current allocation matrix.
- **Need = Max – Allocation.**

Algorithm steps:

- Check if  $\text{request} \leq \text{Need}$  and  $\text{request} \leq \text{Available}$ .
- Pretend to allocate temporarily.
- Run the safety algorithm to check if the system remains safe.
- If safe, grant the request; otherwise, make the process wait.

Although effective, avoidance requires advance knowledge of maximum demands and is not practical for all systems.

## DEPARTMENT OF COMPUTER APPLICATIONS

### Example

Suppose there are 3 resource types and 5 processes.

If process P1 requests 1 unit of R1, 2 of R2, and 2 of R3, the OS uses the Banker's Algorithm to simulate allocation.

If a safe sequence such as {P3, P4, P1, P5, P2} exists after allocation, the request is granted.

If no safe sequence exists, P1 must wait until some other process finishes and releases resources.

### Conclusion

Deadlock avoidance dynamically ensures safety by examining resource allocation at runtime.

It offers better concurrency than prevention while maintaining system reliability.

However, it requires prior knowledge of maximum demands and may introduce computational overhead.

## Topic 11 – Deadlock Detection and Recovery

### Introduction

If neither prevention nor avoidance is applied, the system must rely on **deadlock detection and recovery**.

This approach allows deadlocks to occur but detects them using algorithms that analyze the resource allocation state.

Once detected, the operating system takes recovery actions such as preempting resources or terminating processes.

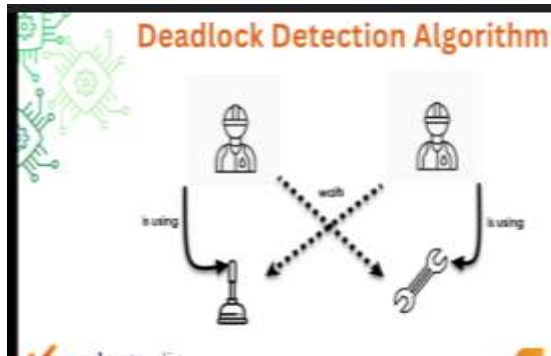
This strategy is suitable for systems where deadlocks are rare but must be resolved when they occur, such as databases and general-purpose operating systems.

The detection mechanism depends on whether resources have single or multiple instances.

The OS periodically runs detection algorithms to identify cycles in the resource allocation graph or deadlock matrices.



## DEPARTMENT OF COMPUTER APPLICATIONS



### Explanation

#### Detection Algorithms:

##### 1. Single Instance per Resource Type:

- Use a Resource Allocation Graph (RAG).
- A cycle in the graph indicates a deadlock.

##### 2. Multiple Instances per Resource Type:

- Maintain data structures similar to Banker's Algorithm:
  - **Available, Allocation, Request.**
- The detection algorithm simulates allocation to check if all processes can complete.
- If some cannot, they are involved in a deadlock.

#### Recovery Techniques:

##### 1. Process Termination:

- Abort one or more processes to break the cycle.
- Criteria for selecting victims: process priority, time executed, resource usage, etc.
- Abort all deadlocked processes or one at a time until deadlock is resolved.

##### 2. Resource Preemption:

- Temporarily take resources from some processes and allocate them to others.
- Must ensure no data corruption occurs during preemption.

## DEPARTMENT OF COMPUTER APPLICATIONS

- The OS may roll back processes to a safe state and restart them later.

Both techniques aim to restore the system to a safe, deadlock-free condition.

### Example

In a system with three processes (P1, P2, P3) and two resources (R1, R2):

- P1 holds R1 and requests R2.
- P2 holds R2 and requests R1.
- P3 waits for R1.

The detection algorithm finds a cycle (P1 → R2 → P2 → R1 → P1).

To recover, the OS may terminate P2 or preempt R2 from P2 and give it to P1.

### Conclusion

Deadlock detection and recovery enable systems to continue operating even when deadlocks occur.

Though it involves overhead, it is practical for systems where deadlocks are infrequent but cannot be tolerated indefinitely.

By carefully detecting and resolving cycles, the OS ensures stability and continuous process execution.